Programming Languages and Techniques (CIS120)

Lecture 19

Feb 27, 2012

Transition to Java

Objects, classes, and interfaces

Announcements

- HW06 Due Thursday, Mar 1 at 11:59:59pm
  - Grace period until Friday, Mar 2

- Note: For the Java portion of the course, we recommend creating a *new* Eclipse workspace
  - So that you don't have to switch settings between OCaml/Java when you move back and forth

Looking Back…

OCaml: What's Left

OCaml is not a very large language — we've actually seen most of its important features. But we've omitted a few…

- Module system
  - One of OCaml's most interesting features is its excellent support for large-scale programming
  - We saw just the tip of the iceberg: *structures* and *signatures*
  - Key feature: *functors* (functions from structures to structures)
- Object system
  - OCaml actually includes a powerful system of classes and objects
  - We left them out to avoid confusion with Java's way of doing things
- Miscellaneous handy type-system features
  - e.g. "polymorphic variants" (used, for example, to support parameter passing by name instead of by position)
  - Type inference – almost all of the type annotations we've been using can be omitted.

Recap: The Functional Style

- Core ideas:
  - value-oriented programming
  - immutable (persistent / declarative) data structures
  - recursion (and iteration)
  - functions as data
  - generic types for flexibility (i.e. ‘a list)
  - abstract types to preserve invariants (i.e. BSTs, queues)
- Good for:
  - simple, elegant descriptions of complex algorithms and/or data
  - parallelism, concurrency, and distribution
  - “symbol processing” programs (compilers, theorem provers, etc.)

Language Support for FP

- "Functional languages" (OCaml, Standard ML, F#, Haskell, Scheme) promote this style as a default and work hard to implement it efficiently
- "Hybrid languages" (Scala, Python) offer it as one possibility among others
- Mainstream "Object Oriented" languages (Java, C#, C++) favor a different style by default
  - But many common OO idioms and *design patterns* have a functional flavor (e.g. the "Visitor" pattern is analogous to transform)
  - And most of them are gradually adding features (like anonymous functions) that make functional-style programming more convenient
  - Best practices discourage use of imperative state

OCaml vs. Java

```ocaml
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let is_empty (t:'a tree) =
  begin match t with
    | Empty -> true
    | Node(_,_,_) -> false
  end

let t : int tree = Node(Empty,3,Empty)
let ans : bool = is_empty t
```

```java
interface Tree<A> {
  public boolean isEmpty();
}
class Empty<A> implements Tree<A> {
  public boolean isEmpty() {
     return true;
  }
}
class Node<A> implements Tree<A> {
  private final A v;
  private final Tree<A> lt;
  private final Tree<A> rt;

  Node(Tree<A> lt, A v, Tree<A> rt) {
    this.lt = lt; this.rt = rt; this.v = v;
  }

  public boolean isEmpty() {
    return false;
  }
}

class Program {
  public static void main(String[] args) {
    Tree<Integer> t =
    new Node<Integer>(new Empty<Integer>(),
     3, new Empty<Integer>());
    boolean ans = t.isEmpty();
  }
}
```

Course Overview

- Declarative programming
  - *persistent* data structures
  - *recursion* is main control structure
  - heavy use of functions as data
- Imperative programming
  - *mutable* data structures (that can be modified “in place”)
  - *iteration* is main control structure
- Object-oriented programming
  - pervasive “abstraction by default”
  - mutable data structures / iteration
  - heavy use of functions (objects) as data

Imperative programming

Java (and C, C++, C#)

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

OCaml

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

OO programming

Java (and C, C++, C#)

- Primitive notion of object creation (classes, with fields, methods and constructors)
- Flexibility through extension:
  Subtyping allows related objects to share a common interface
  (i.e. button <: widget)

OCaml

- Explicitly create objects using a record of higher order functions and hidden state
- Flexibility through composition: objects can only implement one interface
  (i.e. button = widget *
  label_controller *
  notifier_controller).

Looking Forward

Today: A high-level tour of Java.

Objects

from OCaml to Java

"Objects" in OCaml

```
type counter = {inc: unit->int;
                dec: unit->int}

let newcounter () : counter =
  let r = ref 0 in
  {
    inc = (fun () ->
             r := !r + 1; !r);
    dec = (fun () ->
             r := !r - 1; !r);
  }

let c = newcounter()
;; print_int (c.inc())
;; print_newline()
;; print_int (c.dec())
;; print_newline()
```

Why is this an object?

- *Encapsulated local state* only visible to the methods of the object
- Object is *defined by what it can do*—local state does not appear in the interface
- There is a way to *construct* new object values that behave similarly

Critique of Hand-Rolled Objects

- "Roll your own objects" made from records, functions, and references are good for understanding...

```
type counter = {inc: unit->int; dec: unit->int}

let newcounter () : counter =
  let r = ref 0 in
  {
    inc = (fun () -> r := !r + 1; !r);
    dec = (fun () -> r := !r - 1; !r)
  }
```

- ...but not that good for programming
  - minor: syntax is clunky (too many parens, etc.)
  - major: OCaml's record *types* are too rigid, cannot reuse functionality

```
type reset_counter = {inc: unit->int; dec: unit->int;
                      reset : unit -> unit}
```

Java Objects and Classes

- *Object*: a structured collection of *fields* (aka *instance variables*) and *methods*
- *Class*: a template for creating objects
- The class of an object specifies
  - the types and initial values of its local state (fields)
  - the set of operations that can be performed on the object (methods)
  - one or more *constructors*: code that is executed when the object is created (optional)
- Every Java object is an *instance* of some class
- Can (optionally) implement an *interface* that specifies it in terms of its operations

Objects in Java

class declaration

```java
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

class name

instance variable

constructor

methods

object creation and use

```java
public class Main {

  public static void
     main (String[] args) {

       Counter c = new Counter();

       System.out.println( c.inc() );

    }
}
```

constructor invocation

method call

Creating Objects

- *Declare* a variable to hold the `Counter` object
  - Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for `Counter` to create a `Counter` instance with keyword "new" and store it in the variable

```
Counter c;
c = new Counter();
```

- … or declare and initialize together (preferred)

```
Counter c = new Counter();
```

Constructors with Parameters

```
public class Counter {

  private int r;

  public Counter (int r0) {
    r = r0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter(3);

      System.out.println( c.inc() );

   }
}
```

constructor invocation

Creating objects

- Every Java variable is mutable

```
Counter c;
c = new Counter(2);
c = new Counter(4);
```

- A Java variable of *reference* type contains the special value "null" before it is initialized

```
Counter c;
if (c == null) {
  System.out.println ("null pointer");
}
```

Single = for assignment
Double == for equality testing

Using objects

- At any time, a Java variable of reference type can contain either the special value “null” or a pointer into the heap
  - i.e., a Java variable of reference type "T" is like an OCaml variable of type "T option ref"
  - The dereferencing of the pointer and the check for “null” are implicitly performed every time a variable is used

```
let f (co : counter option ref) : int =
  begin match !co with
  | None ->
     failwith "NullPointerException"
  | Some c -> c.inc()
  end
```

```
class Foo {
  public int f (Counter c) {
    return c.inc();
  }
}
```

- If null value is used as an object (i.e. with a method call) then a NullPointerException occurs

Encapsulating local state

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

r is *private*

constructor and methods can refer to r

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

      System.out.println( c.inc() );

   }
}
```

other parts of the program can only access public members

method call

Encapsulating local state

- Visibility modifiers make the state local by controlling access
- Basically:
  - public : accessible from anywhere in the program
  - private : only accessible inside the class
- Design pattern: first cut
  - Make *all* fields private
  - Make constructors and methods public

(There are a couple of other protection levels — protected and "package protected". The details are not important at this point.)

Interfaces

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Describes a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {
  public int getX();
  public int getY();
  public void move(int dx, int dy);
}
```

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface
- That class fulfills the contract implicit in the interface

```
public class Point implements Displaceable {
  private int x, y;
  public Point(int x0, int y0) {
    x = x0;
    y = y0;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

interfaces implemented

methods required to satisfy contract